Dynamic programming

And how to bruteforce your way into efficiently solving problems

"Once you understand it, dynamic programming is probably the easiest algorithm design to apply in practice. […] until you understand dynamic programming, it seems like magic", S.S. Skiena

How to solve an optimisation problem

Strategy 1 : Exhaustive search

- Guarantied to find the best solution
  → Global optimality
- Really slow : we have to enumerate all possible combinations

Strategy 2 : Greedy algorithm

- Based on local optimality with heuristics: "take the best local decision at each step"
  → No guaranty for global optimality
- Usually efficient

Strategy 3 : Dynamic Programming

What if we could have everything at once?

The concept : Divide & Conquer X Memoization

Divide & Conquer

- Break down the problem into smaller problems
- Solve the subproblems
- Combine the results

Memoization

- Identify the redundant subproblems to solve them only once

A (really) quick reminder on Divide & Conquer

Question : is 9 in this sorted list? 1 2 5 7 8 9 10 12

→ Naive search : look at every element from the start, O(n)

→ Binary search : split the search space in half at each iteration, O(log(n))

1 2 5 7 8 9 10 12

1 2 5 7 8 9 10 12

1 2 5 7 8 9 10 12

1 2 5 7 8 9 10 12 Found it !

How to choose the subproblems?

- The idea is to find how to recursively reach the easy problems, without losing optimality
- We want the result of the subproblems to be useful in order to compute the result of the greater one we're struggling with

Bellman's principle of optimality : if subproblems' result are optimal then their combination will be too

Bellman equation

Bellman equation are the rules to go from one subproblem to another.

- They need to go with Bellman's principle of optimality
- Try smally decreasing any parameter of the problem (number of element, size of a line, ...)
- The subproblems have to strictly decrease in size, leaning toward the base cases

  → There cannot be any cycle !

Don't forget the base cases, The recursion has to stop at some point !

A common example : the fibonacci sequence

Rules (Bellman equation) :

$F(0) = 0$, $F(1) = 1$

$F(n) = F(n - 1) + F(n - 2)$

How to compute efficiently $F(n)$ for a large n?

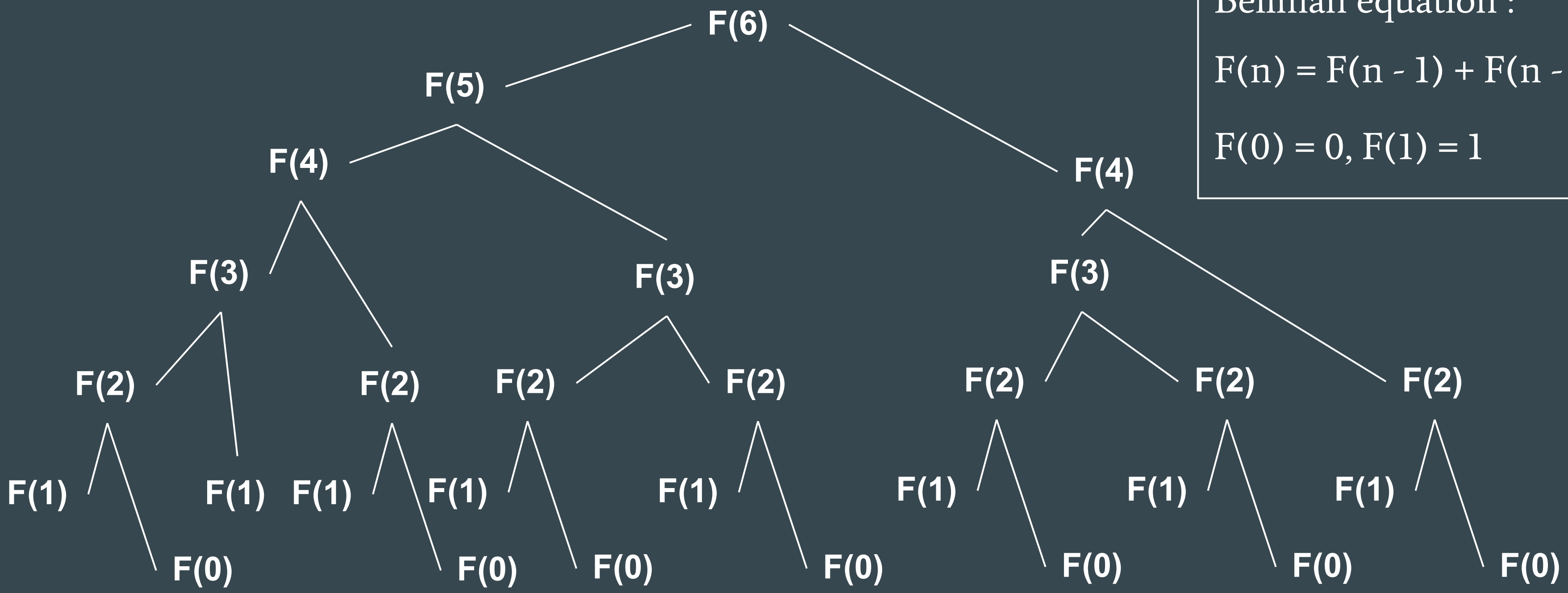
A common example : the fibonacci sequence

Naive recursive algorithm :

```
def Fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return Fibonacci(n - 1) + Fibonacci(n - 2)

Fibonnaci(n)
```

A common example : the fibonacci sequence



Bellman equation :

$F(n) = F(n - 1) + F(n - 2)$

$F(0) = 0, F(1) = 1$

This algorithm runs in $O(2^n)$, there is probably a better way to do this...

Memoization, or how to trade space for time

- Keep track of what subproblem has already been solved
- Keep those subproblems' result in memory
- If one shows up again, take its result out of memory instead of computing it

An efficient algorithm for computing Fibonacci sequence

```
def Fibonacci(n):
      if n == 0:
            return 0
      if n == 1:
            return 1
      if memo[n] != -1:
            return memo[n]
      else:
            result = Fibonacci(n - 1) + Fibonacci(n - 2)
            memo[n] = result
            return result

memo = [-1] * (n + 1)
Fibonnaci(n)
```

An efficient algorithm for computing Fibonacci sequence

```
def Fibonacci(n):
      if n == 0:
            return 0
      if n == 1:
            return 1
      if memo[n] != -1:
            return memo[n]
      else:
            result = Fibonacci(n - 1) + Fibonacci(n - 2)
            memo[n] = result
            return result

memo = [-1] * (n + 1)
Fibonnaci(n)
```

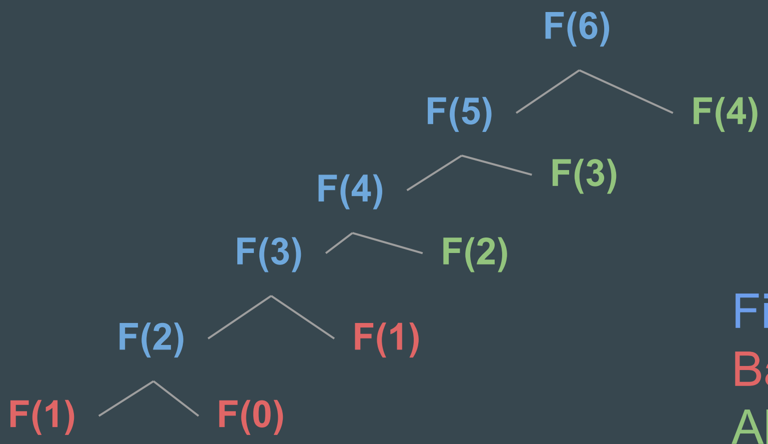
Base cases

Use precomputed result if possible

Bellman equation

Store result in memory !

Sequence of calls with this algorithm



Bellman equation :

$$F(n) = F(n - 1) + F(n - 2)$$

$$F(0) = 0, F(1) = 1$$

First call

Base case

Already computed

We've achieved a O(n) algorithm !

Bottom Up vs Top Down

Bottom up

- Iterative version
- Build subproblems and grow toward the global one
- A bit faster, better for memory concerns

Initial problem

Base cases

Top down

- Recursive version
- Decrease from the initial problem to the base cases
- Easy to implement once you have the Bellman equation

Initial problem

Base cases

Bottom Up vs Top Down : Fibonacci sequence

Bottom up

```
fibo = [0] * (n + 1)
fibo[0] = 0
fibo[1] = 1
for k in range(2, n + 1):
     fibo[k] = fibo[k - 1] + fibo[k - 2]
```

Top down

```
def Fibonacci(n):
     if n == 0:
          return 0
     if n == 1:
          return 1
     if memo[n] != -1:
          return memo[n]
     else:
          result = Fibonacci(n - 1) +
     Fibonacci(n - 2)
          memo[n] = result
          return result

memo = [-1] * (n + 1)
Fibonnaci(n)
```

Quick recap : What do we need ?

- Find a way to split the initial problem into subproblems that are easier to solve
- Find the Bellman equation to jump from one problem to its subproblems easily
- Think about the base cases you can consider !
- Don’t forget to memoize
- Enjoy being a wizard of algorithms!

Next lecture :

- How are you going to store the subproblems’ results? Choose your data structures wisely !
- Common DP problems
- Take a step back : how to quickly evaluate if DP is necessary / will be fast enough?

Credits

Slides: Arthur Tondereau, Louis Sugy for INSAlgo

The algorithm design manual, Steven S.Skiena

Wikipedia of course!